

Nº Orden	Apellido y nombre	L.U.	Cantidad de hojas
13			5

Corrijo: Gonzalo

Organización del Computador 2

Recuperatorio del Primer Parcial — 28/06/18

1 (40)	2 (40)	3 (20)	
40	34	16	90 (A)

Normas generales

- Numere las hojas entregadas. Complete en la primera hoja la cantidad total de hojas entregadas.
- Entregue esta hoja junto al examen, la misma **no** se incluye en la cantidad total de hojas entregadas.
- Está permitido tener los manuales y los apuntes con las listas de instrucciones en el examen. Está prohibido compartir manuales o apuntes entre alumnos durante el examen.
- Cada ejercicio debe realizarse en hojas separadas y numeradas. Debe identificarse cada hoja con nombre, apellido y LU.
- La devolución de los exámenes corregidos es personal. Los pedidos de revisión se realizarán por escrito, antes de retirar el examen corregido del aula.
- Los parciales tienen tres notas: I (Insuficiente): 0 a 59 pts, A- (Aprobado condicional): 60 a 64 pts y A (Aprobado): 65 a 100 pts. No se puede aprobar con A- ambos parciales. Los recuperatorios tienen dos notas: I: 0 a 64 pts y A: 65 a 100 pts.

Ej. 1. (40 puntos)

Sea una lista circular que respeta la siguiente estructura:

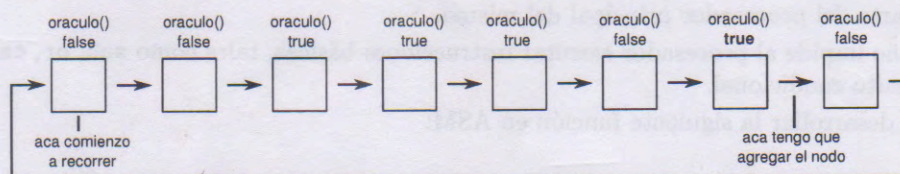
```
struct nodo {
    struct nodo* siguiente,
    int dato,
    bool (*oraculo)()
}
```

Donde `siguiente` es un puntero al siguiente nodo, `dato` es un valor entero almacenado y `oraculo` es un puntero a función que no recibe nada y devuelve un valor de tipo `bool`. `bool` es un entero de 4 bytes, que se interpreta como `false` si vale 0 y `true` en caso contrario.

Se pide escribir 2 funciones:

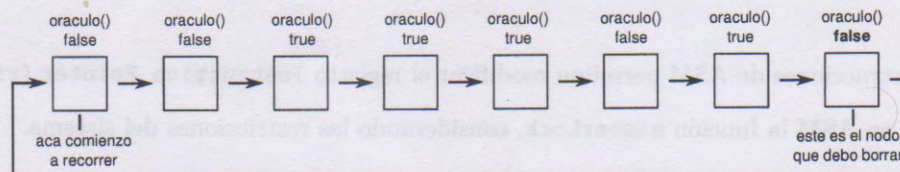
- `void insertarDespuesDelUltimoTrue(nodo* listaCircular, int nuevoDato, bool (*nuevoOraculo)())`:

Inserta un nuevo nodo después del último nodo para el cual el llamado a su `oraculo` devuelva `true`. El nuevo nodo debe contener los campos `dato` y `oraculo` pasados por parámetro. Si el oráculo de ningún nodo devuelve `true`, no se debe insertar nada.



- `void borrarUltimoFalse(nodo** listaCircular)`:

Borra el último nodo cuya llamada a `oraculo` devuelva `false`. De borrar el primero debe modificar el puntero a la lista circular. Si el oráculo de ningún nodo devuelve `false`, no se debe borrar nada.



- (8p) a. Implementar la función `insertarDespuesDelUltimoTrue` en C.
- (15p) b. Implementar la función `insertarDespuesDelUltimoTrue` en ASM.
- (17p) c. Implementar la función `borrarUltimoFalse` en ASM.

Ej. 2. (40 puntos)

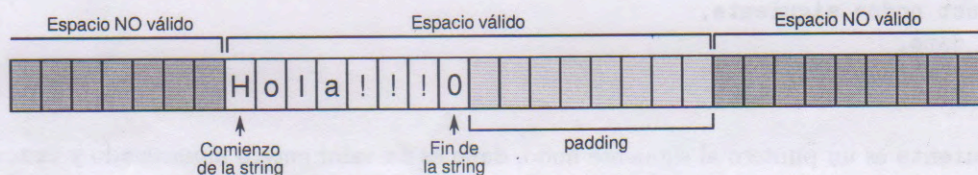
Un string de C es una cadena de caracteres de un byte codificados usando ASCII que termina en 0 (nulo). Por ejemplo, la cadena Hola!!! se codifica:

H	o	l	a	!	!	!	
0x48	0x6f	0x6c	0x61	0x21	0x21	0x21	0x00

Se pide implementar dos funciones:

- **double promedio(char* str):**
Calcula el promedio de los caracteres del string, sin considerar el caracter nulo.
Por ejemplo, para la string anterior: $\frac{0x48+0x6f+0x6c+0x61+0x21+0x21+0x21}{7} = 69,57\dots$
- **void reemplazar(char* str, int longitud, char viejo, char nuevo):**
Toma un string y dos caracteres, y reemplaza cada ocurrencia del primer caracter por el segundo en el string. Por ejemplo, para la string anterior, reemplazamos cada ocurrencia de a por b queda Holb!!!. Si reemplazamos cada ocurrencia de ! por x, queda Holaxxx. Si reemplazamos cada ocurrencia de r por q, el string queda igual.

Nota: La longitud de la string puede ser arbitrariamente grande, considerar que el área ocupada por la string es múltiplo de 16 bytes, es decir, se agrega *padding* al final de la misma que puede ser leído y modificado.



(20p) a. Implementar, usando SIMD, la función promedio.

(20p) b. Implementar, usando SIMD, la función reemplazar.

Ej. 3. (20 puntos)

Los desarrolladores del software que ejecuta un satélite están muy seguros que la radiación cósmica destruye parte del procesador principal del mismo.

Este daño impide al procesador ejecutar instrucciones básicas, tales como `sub`, `or`, `call`, `loop`, `jmp` y ni ningún salto condicional.

Se pide desarrollar la siguiente función en ASM:

```
int masterLock( int (*connectionRelease)(int), int (*magneticToroid)(int), int value ) {
    int response = connectionRelease(value);
    response = response + magneticToroid(value);
    return (67 - response);
}
```

(5p) a. ¿Qué instrucciones de ASM permiten modificar el registro Instruction Pointer (rip)?

(15p) b. Escribir en ASM la función `masterLock`, considerando las restricciones del sistema.

Nota: Recordar que el inverso aditivo de un número en complemento a dos es: $\text{notBitAbit}(n) + 1$

Ejercicio 1

a) La función debería hacer básicamente lo siguiente:

- Guardar un puntero al nodo por el que empezó a recorrer, así sabe cuándo terminó.
- Guardar un puntero al último true, que se inicializará en NULL.
- Ciclar hasta volver a encontrarse con el primer nodo, actualizando el puntero al último true cada vez que oráculo devuelva true. Si no se encontró ninguno, el puntero se mantiene en NULL.
- Si el puntero es NULL, no hay que hacer nada más. Si no lo es, hay que agregar un nodo nuevo como siguiente del puntero; su siguiente debe ser el siguiente de ese último true.

~~void insertarDespuesDelUltimoTrue~~

```
void insertarDespuesDelUltimoTrue(nodo* listaCircular, int nuevoDato,
    bool (*nuevoOraculo)()) {
```

// Suponemos que la lista tiene al menos un nodo
// (añadido en clase).

```
nodo* primero = listaCircular, *actual = listaCircular;
```

```
nodo* ultimoTrue = NULL; oraculo
```

```
bool resultado = (actual->nuevoOraculooraculo)();
```

```
if (resultado) ultimoTrue = actual;
```

```
actual = actual->siguiente;
```

```
while (actual != primero) { oraculo
    resultado = (actual->nuevoOraculooraculo)(); ✓
```

```
if (resultado) ultimoTrue = actual;
```

```
actual = actual->siguiente;
```

```
}
```

```
if (ultimoTrue) {
```

```
nodo* nuevo = malloc(sizeof(nodo));
```

```
nuevo->siguiente = ultimoTrue->siguiente;
```

```
nuevo->dato = dato;
```

```
nuevo->oraculo = nuevoOraculo;
```

```
ultimoTrue->siguiente = nuevo; ✓
```

```
}
```

```
}
```

Offsets para usar en ASM:

+0	siguiente	1 byte
+8	dato	4 bytes
+16	oraculo	4 bytes

Perfecto!

Alineados a 8 porque la estructura no es packed.

```
%define OFFSET_SIGUIENTE 0
```

```
%define TAM_NODO 24
```

```
%define OFFSET_DATO 8
```

```
%define NULL 0
```

```
%define OFFSET_ORACULO 16
```

NOTA

b) Implementación en ASM
extern malloc

insertarDespuesDelUltimoTrue:

ultimoTrue

; ABX: primero, R12: actual, R13: ~~ultimoTrue~~
; R14: nuevoDato, R15: nuevoOraculo
; Uso registros intocables según la convención C
; para que las llamadas a oraculo y malloc
; no toquen las ~~variables~~ que uso en el ciclo.

```
push rbp ; A
mov rbp, rsp
push rbx ; D
push r12 ; A
push r13 ; D
push r14 ; A
push r15 ; D
sub rsp, 8 ; pila alineada
```

```
mov rbx, rdi
mov r12, rdi
mov r13, NULL
mov r14, rsi
mov r15, rdx
```

```
call [r12+OFFSET_ORACULO] ; para ver si el primer nodo da true
cmp eax, 0 ; porque bool ocupa 4 bytes
je avanzar.avanzar ; no era true, no cambiamos ultimoTrue
mov r13, r12 ; era true, actualizamos ultimoTrue
```

.avanzar:

```
mov r12, [r12+OFFSET_SIGUIENTE]
```

ciclo:

```
cmp r12, rbx
je .finCiclo ; llegamos de nuevo al primero, listo
call [r12+OFFSET_ORACULO]
cmp eax, 0
je .avanzarCiclo ; actualizamos ultimoTrue
mov r13, r12
```

.avanzarCiclo:

```
mov r12, [r12+OFFSET_SIGUIENTE]
jmp .ciclo
```

Perfecto

.finCiclo:

```
cmp r13, NULL
je .fin ; no hay ningún true
```

```
continuar
mov rdi, TAM_NODO
call malloc ; RAX → nuevo
mov r8, [r13+OFFSET_SIGUIENTE]
mov [rax+OFFSET_SIGUIENTE], r8
mov [rax+OFFSET_DATO], r14 (*)
mov [rax+OFFSET_ORACULO], r15
mov [r13+OFFSET_SIGUIENTE], rax
```

.fin:

```
add rsp, 8
pop r15
pop r14
pop r13
pop r12
pop rbx
ret
```

(*) Esto es válido porque en las direcciones de memoria más bajas irá EAX (el dato). Como en las más altas va fruta, no hay problema con que la parte alta de RAX tenga fruta.

c) borrarUltimoFalse debería seguir una lógica parecida, con algunas salvedades:

- En vez de ~~guardar~~ guardar un puntero al último nodo true, guarda uno al último nodo false
- Guarda el doble puntero inicial (el que le entró como argumento) para modificarlo en caso de que el nodo borrado sea el primero. (Esto en la implementación ASM).
- Guarda un puntero al nodo anterior, así puede saber cuál es el anterior al borrado y reconectar lo que haga falta.

Primero lo hago en C para agilizar la escritura en ASM:

```
void borrarUltimoFalse(nodo** listaCircular) {
    nodo* actual = *listaCircular, *anterior = NULL, *primero = *listaCircular;
    nodo* ultimoFalse = NULL, *anteriorFalse = NULL;
    bool resultado = (actual->oraculo)();
    if (!resultado) ultimoFalse = actual;
    anterior = actual;
    actual = actual->siguiente;

    while (actual != primero) {
        resultado = (actual->oraculo)();
        if (!resultado) {
            ultimoFalse = actual;
            anteriorFalse = anterior;
        }
        anterior = actual;
        actual = actual->siguiente;
    }

    if (ultimoFalse) {
        if (ultimoFalse == primero) { //borro 1ero
            anteriorFalse = anterior; //caso borde, era NULL
            *primero = ultimoFalse->siguiente; //también podría ser el anterior,
            // consulte y me dijeron que
            // esto estaba bien
            anteriorFalse->siguiente = ultimoFalse->siguiente;
            free(ultimoFalse);
            if (actual->siguiente == actual)
                *listaCircular = NULL; //caso borde, único elemento
        }
    }
}
```

Se podía hacer un poco más elegante con un do { while (...); }

Esto va DESPUÉS del if() {} porque si la lista tiene un único elemento y hay algún false, ese elemento es el primero y no es (el mismo que actual) y no es válido hacer actual->siguiente cuando esa memoria ya fue liberada. (gracias por hacerlo en C)

Aclaración: en realidad el caso en el que tiene un único elemento está incluido dentro del caso en el que se está borrando el primero. Si no lo sobrescribiera con NULL, tendría un puntero a basura.

extern ~~void~~ free

borrarUltimoFalse:

- ; RBX: primero, R12: actual, R13: anterior, R14: ultimoFalse,
; R15: anteriorFalse, ~~R15: ultimoFalse~~ : listaCircular
[RSP]

```
push rbp ;A
mov rbp, rsp
push rbx ;D
push r12 ;A
push r13 ;D
push r14 ;A
push r15 ;D
push rdi ;A
mov rbx, [rdi]
mov r12, rbx
mov r13, NULL
mov r14, NULL
mov r15, NULL
call [r12+OFFSET_ORACULO]
cmp eax, 0
jne .avanzar
```

mov r14, r12 ; el actual de false ✓

.avanzar:

```
mov r13, r12
mov r12, [r12+OFFSET_SIGUIENTE]
```

.ciclo:

```
cmp r12, rbx ; termino de recorrer, volvi al primero ✓
je .finCiclo
call [r12+OFFSET_ORACULO]
cmp eax, 0
jne .avanzarCiclo
mov r14, r12 ; ultimoFalse := actual
mov r15, r13
```

.avanzarCiclo:

```
mov r13, r12
mov r12, [r12+OFFSET_SIGUIENTE] ✓
jmp .ciclo
```

.finCiclo:

```
cmp r14, NULL
je .fin ; no hay false
cmp r14, rbx
jne .noEsPrimero
mov r15, r13 ; caso borde
mov rax, [rsp]
mov rdx, [r14+OFFSET_SIGUIENTE]
mov [rax], rdx ; cambio primero
.noEsPrimero:
```

```
mov rax, [r14+OFFSET_SIGUIENTE] ✓
mov [r15+OFFSET_SIGUIENTE], rax ✓
```

```
mov rax, [rsp]
mov rdx, [r14+OFFSET_SIGUIENTE]
cmp r12, [r12+OFFSET_SIGUIENTE]
jmp .ciclo .finCiclo
```

.borrar

continúa →

mov [rax], NULL ; cambio primero
; por NULL

.borrar:

```
mov rdi, r12
call free
```

.fin:

```
pop rdi
pop r15
pop r14
pop r13
pop r12
pop rbx
pop rbp
ret
```


Ejercicio 2

- a) Primero calcularé el largo del string con instrucciones de enteros (consulté durante el examen y me dijeron que está permitido).
- La cantidad de veces que tengo que ciclar es $\text{Largo}/16 + 1$ (*) ya que en cada lectura a un XMM entran 16 caracteres y hay una lectura en la que van a entrar los que sobran.
 - Si el largo es múltiplo de 16, debe ciclar $\text{largo}/16$ veces.
 - (*) Solo si el largo no es múltiplo de 16!
 - Para calcular el promedio, puedo usar PHADDW y PHADDQ, pero antes tengo que desempaquetar.

Igual con jle no importa (ver código)

promedio:

~~mov~~ XMM0: suma total, XMM1: suma parcial,
RCX: largo

```

mov     rbp, rsp
push    rbp
mov     rbp, rsp
xor     rcx, rcx
mov     rdi, rdi
largo:
    cmp     byte [rdi], NULL
    je      finLargo
    inc     rcx
    jmp     largo

```

finLargo:

```

pxor     xmm0, xmm0
pxor     xmm2, xmm2
mov     rcx, rcx

```

ciclo: *

```

movdqu   xmm1, [rdi] ; |ch15|...|ch0|, un byte cada uno
movdqa   xmm3, xmm1
punpcklbw xmm1, xmm2 ; |ch1|...|ch0|, un word cada uno
punpcklbw xmm3, xmm2 ; |ch15|...|ch8|
phaddw   xmm1, xmm3 ; |ch15+ch14|...|ch1+ch0|, entran porque 0 ≤ ch15 < 0xFF
phaddw   xmm1, xmm2 ; |0|...|ch15+...+ch12|...|ch3+...+ch0| (*)
phaddw   xmm1, xmm2 ; |0|...|0| |ch15+...+ch8| |ch7+...+ch0| (words)
punpcklwd xmm1, xmm2 ; |0| |0| |ch15+...+ch8| |ch7+...+ch0|
phaddq   xmm1, xmm2 ; |0| |0| |0| |ch15+...+ch0|
cvtdq2pd xmm1, xmm1 ; lo mismo pero como double (*)
addpd    xmm0, xmm1 ; acumula con la suma de los demás chars en XMM1
sub      rcx, 16
jmp      ciclo

```

faltó leer rdi, [rdi+16] entre estas dos líneas

Continúa en la carilla siguiente

(*) La suma de ocho chars seguro entra en un word, ya que el máximo valor posible es $0xFF \cdot 8 = 0xFF00$
Por la misma razón, la suma de 16 entra en un dword

~~Algunas aclaraciones:~~

~~Algunas aclaraciones:~~

.fin:
pinsrb xmm2, rax, 0x0 ; | 0 | largo |

cvtdq2ps xmm2, xmm2
divpd xmm0, xmm2

; esto funciona porque cada double está en el
; quadword más bajo del reg. y la multipli-
; cación es vertical

pop rbp
ret

Algunas aclaraciones:

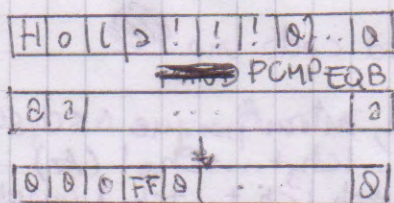
El ejercicio está perfecto, pero como la longitud de los
strings es "arbitraria", había que acumular en punto flotante, no en enteros

- Por más que la suma de los chars en (*) era ~~arbitraria~~ tratada como sword por las operaciones anteriores, se la puede tratar como quadword para pasarla a double, ya que el resto de los bits del quadword más bajo del registro están en 0.
- Aunque sumo con addpd, que es una instrucción con paralelismo, la convención C indica que los valores de tipo double se deben retomar en la parte baja de xmm0, y esto es exactamente lo que está haciendo, no importa cuántos valores use la operación.

b) La función debería hacer lo siguiente:

- Copiar el caracter nuevo y el caracter viejo 16 veces a registros XMM con PINSRB y PSHUFB (con la máscara adecuada).
- Comparar los caracteres del string con el viejo para armar una máscara de bits y así limpiar los que son iguales al viejo en la porción del string y los que ~~seus~~ están en posiciones donde NO son iguales al viejo en las 16 copias del nuevo (todo con PAND y negando la máscara).

Por ejemplo,



- Unir las dos cosas con POR. Así se reemplazan los caracteres.
- ~~Quitar el padding en 0, así que en la última~~

Importante: como el padding NO será sobrescrito (asumimos que nadie va a querer reemplazar el caracter NULL), podemos usar la misma técnica que en el inciso anterior donde comparamos haciendo !e .fin, así no hay que manejar casos borde cuando el largo no es divisible por 16.

.data

ALIGN 16

SHUF_MASK: TIMES 16 DB 0x0

NEGAR: TIMES 16 DB 0xFF

de la posición

copia el byte 0 a todas las del XMM.

para negar con PXOR

.text

reemplazar: ; XMM0: chars originales, XMM1: viejo, XMM2: nuevo

; input: RDI: char* str RSI: int longitud, #

; RDX: char viejo, RCX: char nuevo

push rbp

mov rbp, rsp

~~mov rbp, rsp~~

pinsrb xmm1, dl, 0; | 0 | 0 | ... | 0 | viejo bytes

pshufb xmm1, xmm1, [SHUF_MASK]; | viejo | ... | viejo | bytes

pinsrb xmm2, cl, 0; | 0 | 0 | ... | 0 | nuevo bytes

pshufb xmm2, xmm2, [SHUF_MASK]; | nuevo | ... | nuevo | bytes ✓

movdqa xmm15, [NEGAR] ✓

ciclo:

cmp rsi, 0 *

jle fin

movdqa xmm0, [rdi]

movdqa xmm3, xmm1

pcompb xmm3, xmm0

; xmm3: máscara con 0xFF en las posiciones
; a sobrescribir con el char nuevo

~~movdqa~~

movdqa xmm4, xmm2

pand xmm4, xmm3

; limpia las que no deben ~~scribirse~~ de xmm
; sobrescribir las

pxor xmm3, xmm15

; limpia la máscara

pand xmm0, xmm3

; limpia las que deben sobrescribirse

por xmm0, xmm4

; escribe el char nuevo donde corresponde

sub rdi, 16

jmp rdi, rdi+16

movdqa rdi, [rdi+16]

leq rsi, 16

sub rsi, 16

jmp rsi, rsi+16

fin:

pop rbp

ret

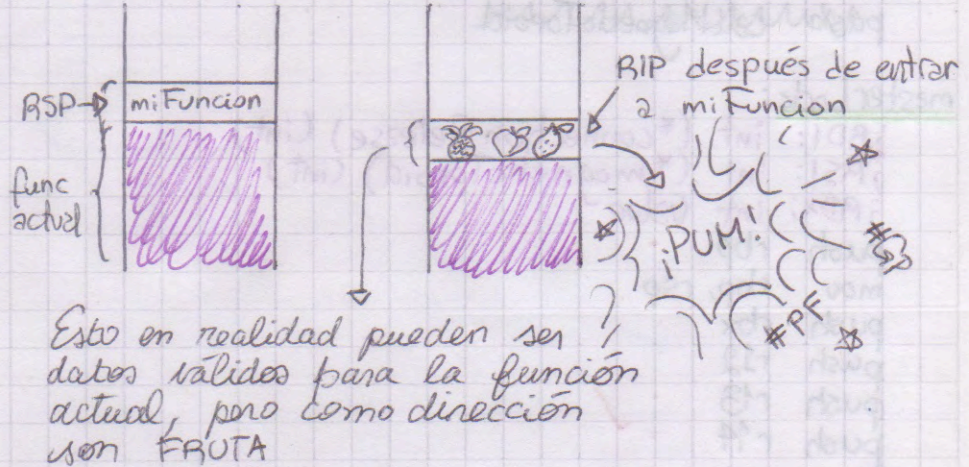
Salir más lindo con pandn

Ab! La longitud del string no es múltiplo de 16 (si no se desea que ocupe, en bytes)

Ejercicio 3

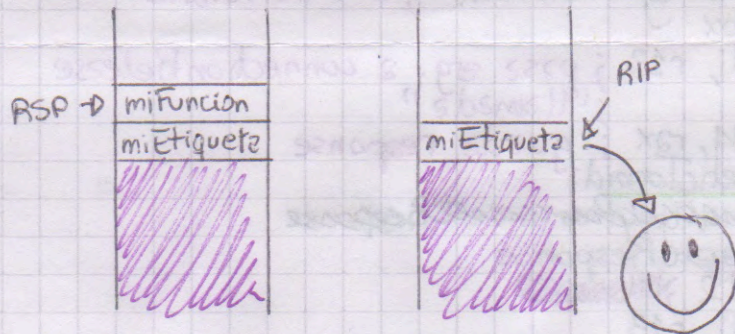
- a) La instrucción `ret` permite modificar el instruction pointer. Por ejemplo, de esta manera puedo saltar a la función que quiero sin usar `call`:

(...)
`push miFuncion`
`ret`
 siguiente instrucción
 (...)



Para que `miFuncion` retorne a la función actual y se ejecute la instrucción siguiente, lo que vea `miFuncion` como dirección de retorno debe ser una etiqueta correspondiente a la instrucción siguiente, de esta forma:

(...)
`push miEtiqueta`
`push miFuncion`
`ret`
~~siguiente~~
`miEtiqueta:`
 siguiente instrucción



buscador:

~~if (RIP == 0) { *connection.Receive(rip); }~~
~~if (RIP == 1) { *connection.Receive(rip); }~~

Continúa en la carilla siguiente

(Pedia que dijeras todas las instrucciones, no solo ret)

guardaToroid:

```
jADI: int (*connectionRelease) (int)  
jRSI: int (*magneticToroid) (int)  
jRDX: int value  
push rbp  
mov rbp, rsp  
push rbx  
push r12  
push r13  
push r14  
mov rdx, rdi ; guarda connectionRelease  
mov r12, rsi ; guarda magneticToroid  
mov r13, rdx ; guarda value  
push callMagneticToroid ; dir. de retorno  
push rbx  
mov rdi, r13 ; pssz arg. a connectionRelease  
ret ; "llamada"  
mov r14, rax ; guarda response  
callMagneticToroid:  
push negateResponse  
push r12 ; response  
mov rdi, r13  
ret  
negateResponse:  
add rax, r14 ; response = response + magneticToroid (value)  
not rax  
inc rax, 1  
add rax, 64  
ret
```

masterLock:

```
jADI: int (*connectionRelease) (int)  
jRSI: int (*magneticToroid) (int)  
jRDX: int value  
push rbp  
mov rbp, rsp  
push rbx  
push r12  
push r13  
push r14  
  
mov rdx, rdi ; guarda connectionRelease  
mov r12, rsi ; guarda magneticToroid  
mov r13, rdx ; guarda value  
  
push callMagneticToroid ; dir. de retorno  
push rbx  
mov rdi, r13 ; pssz arg. a connectionRelease  
ret ; "llamada"  
mov r14, rax ; guarda response  
callMagneticToroid:  
push negateResponse  
push r12 ; response  
mov rdi, r13  
ret  
negateResponse:  
add rax, r14 ; response = response + magneticToroid (value)  
not rax  
inc rax, 1  
add rax, 64  
ret
```